User-Role Reachability Analysis of Evolving Administrative Role Based Access Control [⋆]

Mikhail I. Gofman, Ruiqi Luo, and Ping Yang
Dept. of Computer Science, State University of New York at Binghamton, NY 13902, USA

Abstract. Role Based Access Control (RBAC) has been widely used for restricting resource access to only authorized users. Administrative Role Based Access Control (ARBAC) specifies permissions for administrators to change RBAC policies. Due to complex interactions between changes made by different administrators, it is often difficult to comprehend the full effect of ARBAC policies by manual inspection alone. Policy analysis helps administrators detect potential flaws in the policy specification.
Prior work on ARBAC policy analysis considers only static ARBAC policies. In practice, ARBAC policies tend to change over time in order to fix design flaws or to cope with the changing requirements of an organization. Changes to ARBAC policies may invalidate security properties that were previously satisfied. In this paper, we present incremental algorithms for user-role reachability analysis of ARBAC policies, which asks whether a given user can be assigned to given roles by given administrators. Our incremental algorithms determine if a change may affect the analysis result, and if so, use the information of the previous analysis to incrementally update the analysis result. To the best of our knowledge, these are the first known incremental algorithms in literature for ARBAC analysis. Detailed evaluations show that our incremental algorithms outperform the non-incremental algorithm in terms of execution time.

1 Introduction

Role Based Access Control (RBAC) [12] has been widely used for restricting resource access to only authorized users. In large organizations, RBAC policies are often managed by multiple administrators with varying levels of authority. Administrative Role Based Access Control'97 (ARBAC97) [15] specifies permissions for administrators to change RBAC policies.

Correct understanding of ARBAC policies is critical for maintaining the security of underlying systems. However, ARBAC policies in large organizations are usually large and complex. Consequently, it is difficult to comprehend the full effect of such policies by manual inspection alone. Automated policy analysis helps administrators understand the policy and detect potential flaws. A number of analysis techniques have been developed for user-role reachability analysis of ARBAC [16, 18, 8], which asks, given an RBAC policy and an ARBAC policy, a set of administrators A, a target user u, and a set of roles (called the "goal"), is it possible for administrators in A to assign the target user u to roles in the goal? It has been shown that the reachability analysis for ARBAC (with fixed role hierarchy) is PSPACE-complete [16, 8].

[⋆] This work was supported in part by NSF Grant CNS-0855204

Prior work on ARBAC analysis considers static ARBAC policies. In practice, ARBAC policies tend to change over time in order to correct design flaws or to cope with changing requirements of an organization. Changes to ARBAC policies may invalidate security properties that were previously satisfied. Hence, the policies need to be re-analyzed to verify their overall correctness. In this paper, we present algorithms for user-role reachability analysis of evolving ARBAC. Our algorithms use the information of the previous analysis to incrementally update the analysis result. In a predominant majority of cases, the information obtained from the previous analysis can be used to update the result more quickly than a complete re-analysis. In a small minority of cases, a complete re-analysis cannot be avoided. To the best of our knowledge, these are the first known algorithms in literature for analysis of evolving ARBAC policies.

We propose three forward incremental analysis algorithms – IncFwd1, IncFwd2, and LazyInc. These algorithms are based on the forward algorithm developed in [18], which constructs a *reduced transition graph* from an ARBAC policy; the goal is reachable iff it is a subset of some state in the graph. IncFwd1 determines if a change to the policy may affect the analysis result, and if so, performs non-incremental analysis; otherwise, the algorithm simply returns the previous analysis result. IncFwd2 reuses the transition graph constructed in the previous analysis and incrementally updates the graph. If a change to the policy may affect the transition graph, IncFwd2 updates the graph by pruning the states and transitions that are invalidated by the change and adding new states and transitions that are enabled by the change. Our experimental data show that updating the transition graph is faster than reconstructing it from scratch. LazyInc reuses the graph constructed in the previous analysis, but delays the updates to the graph until a change is made that may affect the analysis result. Subsequently, the algorithm updates the graph based on the change and other delayed changes.

We have also developed a backward algorithm for incremental analysis based on the backward algorithm in [18], which consists of two stages. In the first stage, a goal-directed backward search is performed, which constructs a directed graph G_b. In the second stage, a forward search is performed on G_b to determine if the goal is reachable. Our algorithm reuses G_b as well as the result computed in the second stage. If a can_revoke rule (which specifies authority to remove a user from a role) is added or deleted, G_b remains the same and the algorithm simply updates the result of the second stage. If a can_assign rule (which specifies the authority to add a user to a role) is added or deleted, the algorithm determines if the change may affect G_b, and if so, updates G_b as well as the result of the second stage.

We have implemented the incremental analysis algorithms presented in this paper and compared the algorithms against the non-incremental algorithms on a set of randomly generated policies and a set of operations that change policies. The experimental data show that our incremental analysis algorithms outperform the non-incremental algorithms in terms of the execution time.

Organization The rest of the paper is organized as follows. Section 2 provides a brief overview of RBAC, ARBAC, and the user-role reachability analysis algorithms in [18]. Sections 3 and 4 present the incremental forward and backward analysis algorithms, respectively. The experimental results are given in Section 5. The related work is discussed in Section 6 and our concluding remarks appear in Section 7.

2 Preliminaries

2.1 Role Based Access Control

In Role Based Access Control (RBAC), the association of permissions with users is decomposed into two relations: user-role relation and permission-role relation. The user-role relation $(u, r) \in UA$ specifies that user u is a member of role r. The permission-role relation $(r, p) \in PA$ specifies that role r is granted permission p. RBAC also allows to specify a role hierarchy, which is a partial order on the set of roles. For example, role hierarchy relation $r_1 \succeq r_2$ specifies that role r_1 is senior to role r_2.

Administrative Role-Based Access Control'97 (ARBAC97) [15] controls changes to RBAC policies. Authority to assign a user to a role is specified using the can_assign relation. Each can_assign relation is represented as $can\_assign(r_a, c, r_t)$ where r_a is an administrative role, r_t is the target role, and c is the prerequisite condition (or precondition) of the rule. The precondition is a conjunction of literals, where each literal is either r (positive precondition) or $\neg r$ (negative precondition) for some role r. In this paper, we represent the precondition c as $P \wedge \neg N$ where P contains all positive preconditions in c and N contains all negative preconditions in c. Authority to revoke a user from a role is specified using the can_revoke relation. Each can_revoke relation is represented as $can\_revoke(r_a, r_t)$, which specifies that the administrative role r_a has the permission to remove users from the target role r_t.

ARBAC97 requires that administrative roles and regular roles are disjoint, i.e., administrative roles do not appear in the preconditions and target roles of *can_assign* rules, as well as target roles of *can_revoke* rules. This restriction is called the *separate administration restriction* in [18].

2.2 User-Role Reachability Analysis

The user-role reachability analysis problem asks, given an RBAC policy UA_0, an ARBAC policy ψ, a set of administrators A, a target user u_t, and a set of roles (called the "goal"), is it possible for administrators in A to assign the user u_t to roles in the goal, under the restrictions imposed by ψ? Under the separate administration restriction, the problem can be simplified as follows [16]: (1) Because each user's role memberships are controlled independently of other users' role memberships, UA_0 is simplified to be a set of roles assigned to u_t. (2) Because administrative roles and regular roles are disjoint, it suffices to consider only ARBAC rules whose administrative roles are in A or are junior to roles in A. As a result, administrative roles in A can be merged into a single administrative role and all other administrative roles can be eliminated. With the above simplification, the can_assign rule is simplified as $can\_assign(c, r_t)$, the can_revoke rule is simplified as $can\_revoke(r_t)$, and the user-role reachability analysis problem instance can be represented as a tuple $\langle UA_0, \psi, goal \rangle$.

Our incremental algorithms are developed upon two analysis algorithms in [18]. Below, we summarize these two algorithms.

The forward algorithm In the forward algorithm, roles are classified into *negative* and *non-negative* roles, and *positive* and *non-positive* roles. A role is *negative* if it appears negatively in some precondition in the policy; other roles are *non-negative*. A role is *positive* if it appears in the goal or appears positively in some precondition in the policy; other roles are *non-positive*. A role that is both negative and positive is called *mixed*.

Given a reachability analysis problem instance $I = \langle UA_0, \psi, goal \rangle$, the algorithm first performs a slicing transformation, which computes a set of roles that are relevant to the goal, and eliminates irrelevant roles and rules. A role r is relevant to the goal iff (1) there exist $r_1 \in goal$ and $can\_assign(P \wedge \neg N, r_1) \in \psi$ such that $r \in P \cup N$, or (2) there exist a role $r_1 \in goal$, $can\_assign(P \wedge \neg N, r_1) \in \psi$, and $r_2 \in P$ such that r is relevant to r_2. $Rel_+(I)$ and $Rel_-(I)$ are used to denote the set of positive and negative roles relevant to the goal, respectively.

Next, the algorithm constructs a *reduced transition graph* $G(I)$ from I and performs analysis based on $G(I)$. Each state in $G(I)$ is a set of roles assigned to the target user and each transition describes an allowed change to the state defined by the ARBAC policy ψ. A transition either has the form $ua(r)$ which adds role r to the state, or has the form $ur(r)$ which removes role r from the state. The following reductions are applied: (1) Irrelevant roles and revocable non-positive roles are removed from UA_0; the resulting set of roles is represented as $InitRm_I(UA_0)$. (2) Transitions that revoke non-negative roles or add non-positive roles are prohibited because they do not enable any other transitions; (3) Transitions that add non-negative roles or revoke non-positive roles are *invisible*; other transitions are called *visible* transitions. The invisible transitions will not disable any other transitions.

Let $\text{closure}(UA, I)$ be the largest state that is reachable from state UA by performing invisible transitions enabled from UA using rules in ψ. The algorithm constructs $G(I)$ as follows. First, the algorithm computes $\text{closure}(InitRm_I(UA_0), I)$, which is the initial state of $G(I)$. The algorithm then computes states reachable from $\text{closure}(InitRm_I(UA_0), I)$: given a state s, there is a transition $s \overset{ua(r)}{\rightarrow} \text{closure}(s \cup \{r\})$ if there exists $can\_assign(c, r) \in \psi$ such that r is a mixed role, s does not contain r, and s satisfies c. There is a transition $s \overset{ur(r)}{\rightarrow} \text{closure}(s \setminus \{r\})$ if there exists $can\_revoke(r) \in \psi$ such that r is a mixed role, and s contains r. The algorithm returns true iff a state containing all roles in the goal is reached.

The backward algorithm The backward algorithm in [18] comprises two stages. The first stage performs a backward search from the goal and constructs a directed graph G_b. Each node in G_b is a set of roles. There is an edge $V_1 \overset{\langle P \wedge \neg N, T \rangle}{\rightarrow} V_2$ in G_b if there exists a rule $can\_assign(P \wedge \neg N, T) \in \psi$ such that starting from V_1, revoking all roles that appear in N and adding T, results in node V_2. The second stage performs a forward search from the initial nodes in G_b to determine if the goal is reachable. A node V is an initial node in G_b if $V \subseteq UA_0$. The goal is reachable if there exists a *feasible plan* in G_b. A plan corresponds to a path of edges in G_b from the initial node to the node containing the goal, which is constructed as follows: starting from the initial state, for each edge $V_1 \overset{\langle P \wedge \neg N, T \rangle}{\rightarrow} V_2$ in G_b, a plan contains a transition $s_1 \overset{\alpha}{\rightarrow} s_2$ where s_1 is a state corresponding to V_1, and α is a sequence of actions that revokes roles in $s_1 \cap N$ and adds T. A plan is feasible if it does not revoke irrevocable roles.

To correctly check if G_b contains a feasible plan, each node in G_b is associated with a set of *additional irrevocable roles* ($airs$). $airs(V)$ represents a set of irrevocable roles that appear in the state corresponding to node V, but not in V itself. Formally, given an edge $V_1 \overset{\langle P \wedge \neg N, T \rangle}{\rightarrow} V_2$, $airs(V_2)$ is defined as:

$$\{S \cup ((UA_1 \setminus UA_2) \cap Irrev) \mid S \in airs(UA_1), ((UA_1 \cap Irrev) \cup S) \cap N = \emptyset\}$$

$Irrev$ is the set of irrevocable roles in the policy. $((UA_1 \cap Irrev) \cup S) \cap N = \emptyset$ is the condition that needs to be satisfied in order to add T to the state. The goal is reachable if and only if $airs(goal) \neq \emptyset$.

3 Incremental Forward Algorithms

This section presents three forward algorithms – IncFwd1, IncFwd2, and LazyInc – for incremental reachability analysis of ARBAC. We consider the following changes: (1) add a can_assign rule, (2) delete a can_assign rule, (3) add a can_revoke rule, and (4) delete a can_revoke rule. Because ARBAC with role hierarchy can be transformed to ARBAC without role hierarchy, this paper considers ARBAC without role hierarchy.

IncFwd1 determines if a change may affect the analysis result, and if so, performs non-incremental analysis; otherwise, IncFwd1 returns the previous analysis result. IncFwd2 reuses the transition graph constructed in the previous analysis and incrementally updates the graph. LazyInc also reuses the graph constructed in the previous analysis, but it does not update the graph until an operation that may affect the analysis result is performed. IncFwd1 does not require additional disk space. IncFwd2 and LazyInc require to store the transition graph computed, but are faster than IncFwd1. All three algorithms have the same worst-case complexity as the non-incremental algorithm.

Let $I = \langle UA_0, \psi, goal \rangle$ be a user-role reachability analysis problem instance and $G(I)$ be the transition graph constructed from I using the non-incremental forward algorithm in [18]. Below, we describe the three incremental analysis algorithms in detail.

3.1 Incremental Algorithm: IncFwd1

IncFwd1 is developed based on the following two observations:

- If the analysis result of I is true, then the following changes will not affect the analysis result: (1) add a can_assign rule; (2) add a can_revoke rule; (3) delete a can_assign rule whose target role is non-positive or is irrelevant to the goal; and (4) delete a can_revoke rule whose target role is neither a mixed role nor a negative role in the initial state.
- If the analysis result of I is false, then the following changes will not affect the result: (1) delete a can_assign rule; (2) delete a can_revoke rule; (3) add a can_assign rule whose target role is non-positive or is irrelevant to the goal; and (4) add a can_revoke rule whose target role is neither a mixed role nor a negative role in the initial state.

IncFwd1 uses the slicing transformation result of the previous analysis to determine if a change may affect the analysis result. If so, IncFwd1 performs re-analysis using the non-incremental algorithm; otherwise, IncFwd1 incrementally updates the slicing result and returns the previous analysis result.

3.2 Incremental Algorithm: IncFwd2

IncFwd2 reuses the slicing transformation result and the transition graph computed in the previous analysis, and incrementally updates the graph. States whose outgoing transitions are not computed or not computed completely are marked as "UnProcessed". If a goal that is reachable in I becomes unreachable after a change is made to the policy, IncFwd2 performs non-incremental analysis from states that were previously marked as

```
1  $init' = closure(InitRm_{I_1}(UA_0), I_1)$
2  if $goal \subseteq init'$ then $add\ init'\ to\ G_{inc}(I_1)$;  return $true$ endif
3  if $T \in Rel_+(I_1) \cap Rel_-(I_1)$
4    $W = reached = \{init\}$
5    while $W \neq \emptyset$ do
6      $Remove\ s\ from\ W$
7      for $s \xrightarrow{\alpha} s_1 \in G(I)$ do
8        $add\ s \xrightarrow{\alpha} s_1\ to\ G_{inc}(I_1)$
9        if $goal \subseteq s_1$ then $markUnproc(W \cup \{s\})$; return $true$ endif
10       if $s_1 \notin reached$ then $W = W \cup \{s_1\}$;  $reached = reached \cup \{s_1\}$ endif
11     endfor
12     if $T \in s$ then
13       $s' = closure(s \setminus \{T\}, I_1)$
14       if $s' \notin G(I)$ then $markUnproc(\{s'\})$ endif
15       $add\ s \xrightarrow{ur(T)} s'\ to\ G_{inc}(I_1)$
16       if $goal \subseteq s'$ then $markUnproc(W)$; return $true$ endif
17     endif
18   endwhile
19 elseif $T \in (UA_0 \cap Rel_-(I_1))$ and $T \notin Rel_+(I_1)$ then
20   $W_1 = reached = \{\langle init, init' \rangle\}$
21   while $W_1 \neq \emptyset$ do
22     $Remove\ \langle s, s' \rangle\ from\ W_1$
23     for $(s \xrightarrow{\alpha} s_1) \in G(I)$ do
24       $s'_1 = closure((s_1 \setminus \{T\}) \cup (s' \setminus s), I_1)$
25       $add\ s' \xrightarrow{\alpha} s'_1\ to\ G_{inc}(I_1)$
26       if $goal \subseteq s'_1$ then $markUnproc(\{s'_i | \langle s_i, s'_i \rangle \in W_1\} \cup \{s'\})$;  return $true$ endif
27       if $\langle s_1, s'_1 \rangle \notin\ reached_1$ then $W_1 = W_1 \cup \{\langle s_1, s'_1 \rangle\}$; $reached_1 = reached_1 \cup \{\langle s_1, s'_1 \rangle\}$  endif
28     endfor
29     if $addNewTrans(s, s') == true$ then $markUnproc(\{s'_i | \langle s_i, s'_i \rangle \in W_1\} \cup \{s'\})$; return $true$ endif
30   endwhile
31 else $return\ the\ analysis\ result\ of\ I$ endif
32 if $G_{inc}(I_1) == \emptyset$ then $add\ init'\ to\ G_{inc}(I_1)$;  return $false$ endif
33 $process\ all\ states\ marked\ UnProcessed\ with\ non-incremental\ alg.$
```

Fig. 1. Pseudocode for adding $can\_revoke(T)$

"UnProcessed". Let $G_{inc}(I)$ denote the transition graph computed by IncFwd2 for the problem instance I. Below, we describe IncFwd2 in detail.

Add a can_revoke rule Suppose that $can\_revoke(T)$ is added to the policy ψ. Let $I_1 = \langle UA_0, \psi \cup \{can\_revoke(T)\}, goal \rangle$. Figure 1 gives the pseudocode for constructing graph $G_{inc}(I_1)$ from $G(I)$.

Adding a *can_revoke* rule does not change the result of the slicing transformation. Thus, $Rel_+(I_1) = Rel_+(I)$ and $Rel_-(I_1) = Rel_-(I)$. If T is a mixed role relevant to the goal, the algorithm starts from the initial state $init$ of $G(I)$ and, for every state containing T, adds a transition $ur(T)$ and marks new target state as "UnProcessed" (lines 5 – 18). Otherwise, if T is in UA_0 and is both negative and nonpositive, the algorithm replaces $init$ with $closure(init \setminus \{T\}, I_1)$ and propagates roles in $closure(init \setminus \{T\}, I_1) \setminus init$ to states reachable from $init$ (lines 19–28). Because different states in $G_{inc}(I_1)$ may be computed from the same state in $G(I)$, the workset W_1 contains pairs of the form $\langle s, s' \rangle$ where $s \in G(I)$, $s' \in G_{inc}(I_1)$, and s' is computed from s. The algorithm then calls function *addNewTrans* to add new transitions enabled by the rule; this function returns true if a state containing the goal is reached (line 29). The above process is then repeated on states reachable from $init$. In other cases, the transition graph as well as the analysis result remain the same (line 31).

If the goal is reachable, the algorithm calls function *markUnproc* to mark states in $G_{inc}(I_1)$, whose outgoing transitions are not computed (i.e., the remaining states in W or W_1) or not computed completely (i.e., the state from which the goal is reached by a transition), as "UnProcessed" (lines 9, 16, 26, 29). Otherwise, the algorithm processes all states marked as "UnProcessed" using the non-incremental algorithm (line 33).

Delete a can_revoke rule Suppose that $can\_revoke(T)$ is removed from ψ. Let $I_2 = \langle UA_0, \psi \setminus \{can\_revoke(T)\}, goal\rangle$. Deleting this rule does not change the analysis result if T is not a relevant mixed role and is not a negative role in UA_0. Otherwise, the initial state may change and transitions that revoke T should be deleted. The incremental algorithm is described below.

Deleting a *can_revoke* rule does not change the result of the slicing transformation. If T is a mixed role relevant to the goal, the algorithm starts from the initial state of $G(I)$ and deletes transitions that revoke T. If T is in UA_0 and is both negative and non-positive, then after deleting $can\_revoke(T)$, T should be added back to the initial state. In this case, the algorithm computes a set R_T of roles that may be invalidated by T. A role $r \in R_T$ if (1) T is a negative precondition of a can_assign rule whose target role is r or (2) there exists a role $r' \in R_T$ such that r' is a positive precondition of a can_assign rule whose target role is r. If the number of relevant roles in R_T, which are both positive and non-negative, is small, then for every transition $s \xrightarrow{\alpha} s_1$ in $G(I)$ and state $s' \in G_{inc}(I_2)$ computed from s, the algorithm computes transition $s' \xrightarrow{\alpha} s'_1$ by removing such roles that do not appear in s' from s_1, and computing the closure of the resulting state. The algorithm also checks if transitions that add mixed roles in R_T are invalidated, and if so, removes the transitions. Otherwise, the algorithm performs non-incremental analysis as removing a large number of roles from every state may be more expensive than a complete re-analysis. If the state containing the goal is deleted, the algorithm performs non-incremental analysis from states marked as "UnProcessed".

Add a can_assign rule Suppose that $can\_assign(P \wedge \neg N, T)$ is added to the policy ψ. Let $I_3 = \langle UA_0, \psi \cup \{can\_assign(P \wedge \neg N, T)\}, goal\rangle$. Figure 2 gives the pseudocode for constructing graph $G_{inc}(I_3)$ from $G(I)$.

If T is non-positive or is irrelevant to the goal, then adding this rule does not change the transition graph (line 1). Otherwise, the classification of roles may change: (1) An irrelevant role in I may become relevant in I_3; (2) A relevant role that is both positive and non-negative in I may become a mixed role in I_3; and (3) A relevant role that is both negative and non-positive in I may become a mixed role in I_3. In this case, the algorithm performs incremental slicing to compute $Rel_+(I_3)$ and $Rel_-(I_3)$ (function $inc\_slicing$ in line 3): $Rel_+(I_3) = Rel_+(I) \cup \{r|r$ is a positive role relevant to $T\}$ and $Rel_-(I_3) = Rel_-(I) \cup \{r|r$ is a negative role relevant to $T\}$. The algorithm also computes a set $RelRule$ of rules, which are sufficient to consider during the incremental analysis. Let ρ be a can_assign rule, $target(\rho)$ be the target role of ρ, and $poscond(\rho)$ be the set of positive preconditions of ρ. $RelRule$ is defined as follows:

(1) $can\_assign(P \wedge \neg N, T) \in RelRule$.
(2) a can_assign rule $\rho \in RelRule$ if
  (a) $target(\rho) \in Rel_+(I_3)$ and there exists $\rho' \in RelRule$ such that $target(\rho') \in poscond(\rho)$ or
  (b) $target(\rho)$ is a positive role relevant to T.
(3) $can\_revoke(r) \in RelRule$ if r is a mixed role in I_3 or is a negative role in UA_0.

```
1  if $T \notin Rel_+(I)$ then $return\ the\ analysis\ result\ of\ I$
2  else
3    $\langle Rel_+(I_3), Rel_-(I_3)\rangle = inc\_slicing()$
4    $init' = closure(InitRm_{I_3}(UA_0), I_3)$
5    if $goal \subseteq init'$ then $add\ init'\ to\ G_{inc}(I_3)$; return $true$ endif
6    $RevRoles = ((init' \setminus init) \cap (Rel_+(I_3) \cap Rel_-(I_3)) \setminus Irrev)$
7    $PosNonnegToMix = ((Rel_+(I) \setminus Rel_-(I)) \cap (Rel_+(I_3) \cap Rel_-(I_3)))$
8    $AddRoles = init \cap PosNonnegToMix$
9    if $AddRoles \cup RevRoles \neq \emptyset$ then
10     $\langle answer, lastState\rangle = addTransSeq(init, init', RevRoles, AddRoles)$
11     if $answer == true$ then return $true$ else $W = reached = \{\langle init, lastState\rangle\}$ endif
12   else $W = reached = \{\langle init, init'\rangle\}$ endif
13   while $W \neq \emptyset$ do
14     $Remove\ \langle s, s'\rangle\ from\ W$
15     for $s \xrightarrow{\alpha} s_1 \in G(I)$ do
16       $AddRoles = s_1 \cap PosNonnegToMix$
17       if $AddRoles \neq \emptyset$ then
18         $\langle answer, s_1'\rangle = addTransSeq(s, s', \emptyset, AddRoles)$
19         if $answer == true$ then $markUnproc(\{s_j' | \langle s_j, s_j'\rangle \in W\} \cup \{s'\})$; return $true$ endif
20       else
21         $s_1' = closure(s_1 \cup (s' \setminus s), I_3)$
22         $add\ s' \xrightarrow{\alpha} s_1'\ to\ G_{inc}(I_3)$
23         if $goal \subseteq s_1'$ then $markUnproc(\{s_j' | \langle s_j, s_j'\rangle \in W\} \cup \{s'\})$; return $true$ endif
24       endif
25       if $\langle s_1, s_1'\rangle \notin reached$ then $reached = reached \cup \{\langle s_1, s_1'\rangle\}$; $W = W \cup \{\langle s_1, s_1'\rangle\}$ endif
26     endfor
27     if $addNewTrans(s') == true$ then $markUnproc(\{s_j' | \langle s_j, s_j'\rangle \in W\} \cup \{s'\})$; return $true$ endif
28   endwhile
29   if $G_{inc}(I_3) == \emptyset$ then $add\ init'\ to\ G_{inc}(I_3)$; return $false$ endif
30   $process\ all\ states\ marked\ UnProcessed\ with\ non-incremental\ alg.$
31 endif
```

Fig. 2. Pseudocode for adding $can\_assign(P \wedge \neg N, T)$

$RelRule$ consists of (1) the new rule, (2a) relevant can_assign rules enabled by the new rule, (2b) can_assign rules that enable the new rule, and (3) can_revoke rules which revoke mixed roles or negative roles in UA_0.

Next, the algorithm computes the new initial state $init' = \text{closure}(InitRm_{I_3}(UA_0), I_3)$ (line 4), which may be different from the initial state $init = \text{closure}(InitRm_I(UA_0), I)$ of $G(I)$. Theorem 1 gives the relationship between $init'$ and $init$, which enables us to reuse $G(I)$ to construct $G_{inc}(I_3)$.

Theorem 1 Let $init = \text{closure}(InitRm_I(UA_0), I)$ and $init' = \text{closure}(InitRm_{I_3}(UA_0), I_3)$. One of the following holds: (1) $init = init'$; (2) $init' = \text{closure}(init, I_3)$; or (3) $init$ is reachable from $init'$ through a sequence of transitions: $init' \xrightarrow{ur(r_1)} s_1 \ldots \xrightarrow{ur(r_n)} s_n \xrightarrow{ua(r_{n+1})} s_{n+1} \ldots s_{m-1} \xrightarrow{ua(r_m)} \text{closure}(init \cup s_{m-1}, I_3)$ where $\{r_1, \ldots, r_n\}$ are revocable mixed roles in $init' \setminus init$ and $\{r_{n+1}, \ldots, r_m\}$ are roles in $init \setminus init'$ that are turned from both positive and non-negative to mixed. □

Case (1) states that the initial state does not change after the rule is added. In Case (2), new roles are added to the initial state, but no roles are turned from both positive and non-negative to mixed. In these two cases, the algorithm adds roles in $init' \setminus init$ to $init$ and updates the graph (lines 20–24). In case (3), some roles in $init$ are turned from both positive and non-negative to mixed ($AddRoles$ in line 8), from irrelevant to relevant, or from revocable non-positive to mixed ($RevRoles$ in line 6). In this case, the algorithm calls function *addTransSeq* (line 10) to add a sequence of transitions from $init'$ to closure$(init \cup s_{m-1}, I_3)$, which revokes roles in $RevRoles$, adds roles in $AddRoles$, and marks new states not containing the goal as "UnProcessed". This function returns $\langle answer, lastState \rangle$ where $answer$ is true if the sequence contains the goal state and is false otherwise, and $lastState$ is the last state of the sequence.

Finally, the algorithm calls function *addNewTrans* to add new transitions from $init'$ (line 27) using rules in $RelRule$ and marks new states as "UnProcessed". The above process is then repeated on states reachable from $init$.

Delete a can_assign rule Suppose that $can\_assign(P \wedge \neg N, T)$ is deleted from policy ψ. Let $I_4 = \langle UA_0, \psi \setminus \{can\_assign(P \wedge \neg N, T)\}, goal \rangle$. If T is not a positive role relevant to the goal, then the transition graph remains the same. Otherwise, role T and roles that are reachable through T may become unreachable. Let A_T be a set of roles that may be reachable through T. A role r is in A_T if: (1) T is a positive precondition of a can_assign rule whose target role is r or (2) there exists a role $r' \in A_T$ such that r' is a positive precondition of a can_assign rule whose target role is r.

Deleting $can\_assign(P \wedge \neg N, T)$ may change the classification of T from mixed to both positive and non-negative. This occurs when targets of all can_assign rules, which contain T in their negative preconditions, become non-positive after the rule is deleted. Similarly, roles other than T may change from mixed to both positive and non-negative (*MixtoNonneg*), from mixed to both negative and non-positive (*MixtoNonpos*), and from relevant to irrelevant (*RevtoIrr*). Below, we describe the algorithm.

First, the algorithm performs slicing and computes the new initial state $init' = closure(InitRm_{I_4}(UA_0), I_4)$. $init'$ may be different from the initial state $init = closure(InitRm_I(UA_0), I)$ of $G(I)$. Theorem 2 gives the relationship between $init$ and $init'$, which enables us to reuse $G(I)$ to construct $G_{inc}(I_4)$.

Theorem 2 Let $init = closure(InitRm_I(UA_0), I)$, $init' = closure(InitRm_{I_4}(UA_0), I_4)$, and $Invalid = (init \setminus init') \cap (A_T \cup \{T\})$. One of the following holds: (1) $init' = init$; (2) $init' = init \setminus (RevtoIrr \cup Invalid \cup \{S \in MixtoNonpos | S \text{ is revocable}\})$; or (3) $G(I)$ contains the following sequence of transitions: $init \overset{ua(r_1)}{\rightarrow} s_1 \ldots s_{n-1} \overset{ua(r_n)}{\rightarrow} (init' \cup (s_{n-1} \cap (RevtoIrr \cup Invalid \cup MixtoNonpos)))$ where $\{r_1, \ldots, r_n\} = (init' \setminus init) \cap MixtoNonneg$. $\square$

Case (1) states that the initial state does not change. In Case (2), $init$ does not contain roles turned from mixed to both positive and non-negative, but may contain roles turned from relevant to irrelevant, revocable roles turned from mixed to non-positive, or roles that cannot be rederived after the can_assign rule is deleted. In this case, the algorithm updates the graph from $init$, removes such roles from $init$ and states reachable from $init$, and removes transitions that add or revoke such roles. In case (3), $init$ contains roles turned from mixed to both positive and non-negative. In this case, the algorithm identifies the state $(init' \cup (s_{n-1} \cap (RevtoIrr \cup Invalid \cup MixtoNonpos)))$ in $G(I)$. The algorithm then updates the graph from this state by removing roles in $(s_{n-1} \cap (RevtoIrr \cup Invalid \cup MixtoNonpos))$ from this state and all reachable states.

The graph is updated as follows. For every transition $s_1 \overset{\alpha}{\rightarrow} s_2$ in $G(I)$, if α adds/revokes a role that is turned from mixed to non-positive or if α can no longer be derived, the algorithm removes the transition. Otherwise, if $s_2 \setminus s_1$ contains T and T cannot be re-derived, the algorithm removes T and all roles that cannot be derived without T from s_2. If α adds a role that is turned from mixed to both positive and non-negative, the algorithm removes the transition and updates the graph using a way similar to (3) of Theorem 2.

3.3 Lazy Incremental Forward Algorithm

This section presents a lazy incremental analysis algorithm that delays updates to the transition graph until an operation, which may affect the analysis result, is performed. Due to space constraints, this section presents only the algorithm for the case where the analysis result of the original policy is true. The case where the analysis result is false is handled similarly. Let $I = \langle UA_0, \psi, goal \rangle$ be a reachability analysis problem instance. The algorithm is described below.

Add a can_assign or a can_revoke rule Adding a can_assign or a can_revoke rule does not affect the analysis result though it may affect the transition graph. In this case, we do not update the graph. Instead, we store the rule in a set $DelayedRule$. This set will be used to update the transition graph when an operation that may affect the analysis result is performed.

Delete a can_assign or a can_revoke rule Assume that $can\_assign(P \wedge \neg N, T)$ is deleted from ψ. If T is not a positive role relevant to the goal, the algorithm returns true. Otherwise, the algorithm performs the following steps.

Let $\psi' = (\psi \backslash \{can\_assign(P \wedge \neg N, T)\}) \cup DelayedRule$ and $I' = \langle UA_0, \psi', goal \rangle$. First, the algorithm computes $Rel_+(I')$ and $Rel_-(I')$. The algorithm then updates the transition graph using the deleted rule and delayed operations that may affect the analysis result after the rule is deleted. Such operations include addition of can_assign rules in $DelayedRule$ whose target roles are in $Rel_+(I')$, and addition of can_revoke rules in $DelayedRule$ that revoke relevant mixed roles or negative roles in UA_0. Finally, the algorithm updates the graph using one of the following two approaches.

In the first approach, we update every state and transition of the graph by performing all operations of IncFwd2 described in sections 3.2 with the following difference: when applying Theorem 1, not all transitions adding roles $r_{n+1}, \ldots, r_m$ may be enabled in I' because some of them may depend on the deleted can_assign rule.

In the second approach, the algorithm first considers the operation that deletes $can\_assign(P \wedge \neg N, T)$ and applies IncFwd2 for deleting can_assign to update the graph. If the analysis result changes from true to false, the algorithm updates the graph using rules in $DelayedRule$. The graph is updated in a way similar to algorithms in Figures 1 and 2, but considering multiple added rules.

The second approach is expected to perform better than the first one if the analysis result does not change after the rule is deleted, and worse otherwise. This is because in the former case, the graph is processed once, but in the latter case, the graph is processed twice. Both approaches are expected to perform better than IncFwd2 where each change is processed individually, because the graph will be processed fewer times when applying these two approaches. Our implementation adopted the first approach.

Deleting a can_revoke rule is handled similarly.

4 Incremental Backward Algorithm

This section presents a backward algorithm for incremental user-role reachability analysis. Similar to IncFwd2, our backward algorithm uses the graph G_b and the airs computed in the previous analysis to incrementally update the result. Ideas used in IncFwd1 and LazyInc are also applicable to the backward algorithm.

To support efficient incremental analysis, we extend the non-incremental algorithm as follows: (1) Prior to analysis, we compute a set of roles relevant to the goal, which enables us to quickly determine if a change to the policy may affect the analysis result. (2) We store the graph as well as the airs computed in a file. The initial nodes are also stored, in the order in which they are processed in the second stage. (3) For every node V, we associate every set of $airs(V)$ with the edge along which the set is computed. This enables us to quickly identify the set of airs computed from a given edge. (4) If an edge is processed in the second stage of the algorithm, the edge is marked 1; otherwise, the edge is marked 0. Let $airs'(V)$ denote the airs of node V computed by the incremental algorithm. Below, we describe the algorithm.

Add a can_revoke rule Assume that *can_revoke*(T) is added to the policy. Graph G_b is unaffected and we simply update the airs of nodes from the first initial node V_0. Let $rm(T, airs(V))$ denote $\{S - \{T\} | S \in airs(V)\}$. $airs'(V)$ is computed as follows:

- $airs'(V_0) = rm(T, airs(V_0))$
- For every edge $V_1 \overset{\langle P \wedge \neg N, r\rangle}{\rightarrow} V_2$, $airs'(V_2)$ is computed as the union of the following sets:
  (1) $rm(T, airs(V_2))$
  (2) $\{S \cup (Irrev \cap (V_2 \setminus V_1)) | S \in airs'(V_1) \setminus rm(T, airs(V_1)),$
  $((Irrev \cap V_1) \cup S) \cap N = \emptyset\}$
  (3) $((T \in N)?\{(S \setminus \{T\}) \cup (Irrev \cap (V_2 \setminus V_1)) | S \in airs(V_1),$
  $T \in S, ((Irrev \cap V_1) \cup (S \setminus \{T\})) \cap N = \emptyset\} : \emptyset)$

(1) contains $airs(V_2)$ with T removed from every set. (2) contains sets of additional irrevocable roles computed from new sets in airs of V_1 along the edge. (3) is computed from sets in $airs(V_1)$ that did not satisfy the negative precondition of the edge because they contained T; since T becomes revocable, they are added to $airs'(V_2)$.

If the goal is not reachable from V_0, we pick up the second initial node and repeat the above process. If the goal is reachable, the algorithm updates the airs of nodes until it encounters an edge marked 0. This is because, after a can_revoke rule is added, the goal that was previously unreachable may become reachable and the goal that was previously reachable may be reached earlier.

Delete a can_revoke rule Suppose that *can_revoke*(T) is deleted from the policy. Graph G_b remains the same and we simply update the airs of nodes. First, starting from the first initial node V_0, the algorithm searches G_b along edges marked 1, for nodes whose airs may change. The airs of a node V may change if: (1) T is in the initial state and V does not contain T; or (2) there is an edge $V' \overset{\alpha}{\rightarrow} V$ such that $T \in V'$ and $T \notin V$. If such a node does not exist, the algorithm returns the previous analysis result. Otherwise, the algorithm updates the airs of the node as well as the airs of all nodes reachable from this node by edges marked 1 as follows: for every edge $e = V_1 \overset{\alpha}{\rightarrow} V_2$, if $airs'(V_1) = airs(V_1)$, then $airs'(V_2) = airs(V_2)$; otherwise, $airs'(V_2)$ is computed by removing sets in $airs(V_2)$ that are computed along e and recomputing airs along e using the non-incremental algorithm. If an edge marked 0 is encountered, the algorithm computes the set of airs along this edge. If the goal is not reachable from V_0, the algorithm picks up another initial node and repeats the above process.

Add a can_assign rule Suppose that $can\_assign(P \wedge \neg N, T)$ is added to the policy. If T is not a positive role relevant to the goal, the algorithm returns the previous analysis result; otherwise, the algorithm incrementally updates graph G_b and the airs of nodes.

In the first stage, starting from nodes that contain T, the algorithm computes all reachable edges enabled by the new rule. For each new edge $V \overset{\alpha}{\rightarrow} V'$, if V is a node in G_b and $airs(V) \neq \emptyset$, V is added to a set *affectedNodes*. Also, all new initial nodes are added to a set *newInit*. The new edges are marked 0, indicating that they have not been processed in the second stage.

If the previous analysis result is true, the algorithm returns true. Otherwise, the algorithm updates the airs of nodes as follows. For every node $V \in affectedNodes$, it updates the airs of nodes reachable from V along new edges until a node containing T is encountered. The algorithm then updates the airs of this node as well as the airs of all nodes reachable from this node: for every edge $V_1 \overset{\alpha}{\rightarrow} V_2$, the algorithm computes $airs'(V_2)$ by adding sets that are computed from $airs'(V_1) \setminus airs(V_1)$ along the edge, to $airs(V_2)$. If $airs(goal) \neq \emptyset$, the algorithm returns true. Otherwise, the algorithm computes the airs of nodes reachable from the new initial nodes in $newInit$.

Delete a can_assign rule Suppose that $can\_assign(P \wedge \neg N, T)$ is deleted from the policy. If T is not a positive role relevant to the goal, the algorithm returns the previous analysis result; otherwise, the algorithm performs the following steps.

First, the algorithm back-chains from the goal and marks the following nodes as valid nodes: (1) the goal node, and (2) for every edge $V_1 \overset{\alpha}{\rightarrow} V_2$, if $\alpha \neq \langle P \wedge \neg N, T\rangle$ and V_2 is valid, then V_1 is valid. Valid nodes are nodes that remain in the graph after the rule is deleted. Next, for every edge $V_1 \overset{\langle P\wedge\neg N,T\rangle}{\rightarrow} V_2$, the algorithm deletes the sets in $airs(V_2)$ that are computed through the edge and adds V_2 to a set L_T. The algorithm then deletes all nodes not marked valid, edges containing at least one such node, and edges of the form $V_1 \overset{\langle P\wedge\neg N,T\rangle}{\rightarrow} V_2$. Finally, the algorithm updates the airs of nodes reachable from nodes in L_T: for every edge $V_1 \overset{\alpha}{\rightarrow} V_2$, $airs'(V_2)$ is computed by removing all sets from $airs(V_2)$ that are computed from $airs(V_1) \setminus airs'(V_1)$. If the goal was previously reachable but $airs'(goal) = \emptyset$, the algorithm computes airs of nodes that have not been processed using the non-incremental algorithm.

Note that, an alternative (and incorrect) approach to detecting invalidated nodes is to back-chain from the goal, delete edges computed through the deleted rule, delete nodes without outgoing edges, and then delete edges that contain deleted nodes. Such an approach will fail if the graph contains cycles: nodes in a cycle may not be reachable from the goal node after the rule is deleted, but still contain outgoing edges.

5 Experimental Results

This section presents experimental results of our incremental analysis algorithms. All reported data were obtained on a 2.5GHz Pentium machine with 4GB RAM.

5.1 Experimental Results: Incremental Forward Analysis Algorithms

We apply the non-incremental and incremental forward algorithms to an ARBAC policy ψ_1 generated using the random policy generation algorithm in [18]. The parameter values (e.g., the percentage of mixed roles) in ψ_1 are similar to those in the university

Operation	$goal_1$					$goal_2$				
	States	Trans	Time(Sec.)			States	Trans	Time(Sec.)		
			NonInc	IncFwd1	IncFwd2			NonInc	IncFwd1	IncFwd2
add can_assign	30568	233298	103.69	0	13.87	33396	440052	236.19	233.38	160.22
delete can_assign	19983	153931	58.07	56.23	13.58	18000	220248	84.38	0	18.2
add can_revoke	20866	159290	60.43	0	2.29	19968	247834	97.8	28.23	13.01
delete can_revoke	20297	154853	58.59	44.09	2.39	18432	224928	86.81	0	3.37

Table 1. Performance results of NonInc, IncFwd1, and IncFwd2 on ψ_1 for goals $goal_1$ and $goal_2$.

Goal	States	Trans	Time(Sec.)			
			NonInc	IncFwd1	IncFwd2	LazyFwd
$goal_1$	30219	240802	110.76	45.75	12.58	6.09
$goal_2$	19176	288142	94.86	34.57	12.41	10.59

Table 2. Performance results of NonInc, IncFwd1, IncFwd2, and LazyInc for $goal_1$ and $goal_2$ on ten sequences of operations.

ARBAC policy developed in [18]. We choose two goals $goal_1$ and $goal_2$ such that $goal_1$ is reachable from the initial state $\emptyset$ and $goal_2$ is unreachable, and the size of transition graphs constructed during analysis is reasonably large. We then randomly generate a set of operations that add rules to ψ_1 or delete rules from ψ_1, and use them to compare the performance of our incremental algorithms against the non-incremental algorithm.

Table 1 compares the execution time of the non-incremental algorithm (NonInc), IncFwd1, and IncFwd2 for goals $goal_1$ and $goal_2$, when a change is made to policy ψ_1. Each data point reported is an average over 32 randomly generated rules, except for the case "add can_revoke": because only 10 roles cannot be revoked in ψ_1, we generate only rules that revoke these 10 roles. Columns "States" and "Trans" give the average number of states and transitions computed using NonInc, respectively.

Results for adding/deleting a can_revoke rule Table 1 shows that, when a can_revoke rule is added, IncFwd2 is 26.39 and 7.52 times faster than NonInc for $goal_1$ and $goal_2$, respectively. When only the 2 rules that revoke mixed roles are considered, IncFwd2 is 5.88 times faster than NonInc for $goal_1$ and 2.17 times faster than NonInc for $goal_2$. When considering both goals, IncFwd2 is 10.34 and 1.85 times faster than IncFwd1 and NonInc, respectively.

When a can_revoke rule is deleted, IncFwd2 is 24.51 and 25.76 times faster than NonInc for $goal_1$ and $goal_2$, respectively. When considering only the 8 rules that revoke mixed roles, IncFwd2 is 6.07 times faster than NonInc for $goal_1$ and 6.41 times faster than NonInc for $goal_2$. When both goals are considered, IncFwd2 is 25.24 and 7.65 times faster than NonInc and IncFwd1, respectively.

Results for adding/deleting a can_assign rule All can_assign rules added/deleted are relevant to the goal. Observe from Table 1 that, when a can_assign rule is added, IncFwd2 is 7.48 and 1.47 times faster than NonInc for $goal_1$ and $goal_2$, respectively. This is because, the size of the transition graph increases less significantly for $goal_1$ than $goal_2$ (160550 vs 440052 transitions). In particular, for one of the 32 rules generated for $goal_2$, the size of the graph constructed after the rule is added is 10 times the size of the graph constructed before the rule is added. As a result, IncFwd2 computes a large number of new states and transitions using the non-incremental algorithm, and hence is only slightly faster than NonInc for this rule (1868sec vs 1971sec). When considering both goals, IncFwd2 is 2 and 1.35 times faster than NonInc and IncFwd1, respectively.

When a can_assign rule is deleted, IncFwd2 is 4.3 and 4.6 times faster than NonInc for goals $goal_1$ and $goal_2$, respectively. When both goals are considered, IncFwd2 is 4.48 and 1.77 times faster than NonInc and IncFwd1, respectively.

Operation	$goal_3$						$goal_4$					
	Nodes	Edges	Time (NonIncBack)		Time (IncBack)		Nodes	Edges	Time (NonIncBack)		Time (IncBack)	
			Stage 1	Stage 2	Stage 1	Stage 2			Stage 1	Stage 2	Stage 1	Stage 2
add can_revoke	37053	269878	24.73	23.82	1.06	2.54	12475	66128	3.37	67.28	0.92	17.32
add can_assign	37112	285768	25.6	68.47	2.92	0	12481	66355	3.36	44.21	0.97	5.31
delete can_revoke	37112	284297	69.89	95.26	0.59	0.32	12481	68199	3.35	91.82	0.15	0.40
delete can_assign	37085	283541	25.39	41.83	4.31	0.22	12476	68071	3.56	36.9	1.28	0.16

Table 3. Performance comparison of NonIncBack and IncBack on ψ_2 for $goal_3$ and $goal_4$.

Results for adding/deleting a sequence of rules Table 2 compares the performance of the non-incremental algorithm and three incremental algorithms on 10 sequences of operations. In every sequence, only the last operation affects the analysis result. The column "Time" gives the average execution time of the algorithms for each operation. The results show that, when analyzing policy ψ_1 with $goal_1$, for each operation, LazyInc is 18.18, 7.51, and 2.07 times faster than NonInc, IncFwd1, and IncFwd2, respectively. When analyzing ψ_1 with $goal_2$, for each operation, LazyInc is 8.96, 3.26, and 1.17 times faster than NonInc, IncFwd1, and IncFwd2, respectively.

Disk space consumption IncFwd1 does not require additional disk space. The amount of disk space used in IncFwd2 and LazyInc to store the transition graph is 29.6 MB and 43.8 MB, respectively. This is because: (1) the size of the graph is large; and (2) we store states for every transition, which results in storing a state multiple times, and the size of the state is usually large. We expect that, by storing each state only once and storing the references to states in each transition, we will be able to significantly reduce the disk space consumption without significantly affecting their performance.

5.2 Experimental Results: Incremental Backward Algorithm

Table 3 compares the execution time of non-incremental and incremental backward algorithms. The column headings "NonIncBack" and "IncBack" refer to the non-incremental and incremental algorithm, respectively. We choose a randomly generated policy ψ_2, two goals $goal_3$ and $goal_4$, and two initial states i_3 and i_4, so that (1) $goal_3$ is reachable from i_3 and $goal_4$ is not reachable from i_4; (2) the size of the graph is reasonably large (269889 and 66097 edges for $goal_3$ and $goal_4$, respectively); and (3) both stages of the algorithm are performed during analysis. The additional amount of disk space used to store the graph is 15.4 MB and 3.7 MB for $goal_3$ and $goal_4$ respectively.

When a can_revoke rule is added or deleted, the graph remains the same. The execution time of the first stage of IncBack refers to the time for loading the graph constructed in the previous analysis. Table 3 shows that, when a can_revoke rule is added, IncBack is 13.49 and 3.87 times faster than NonIncBack for $goal_3$ and $goal_4$, respectively. When a can_revoke rule is deleted, IncBack is 181.48 and 173.04 times faster than NonIncBack for $goal_3$ and $goal_4$, respectively.

When a can_assign rule is added to the policy, IncBack is 32.22 times and 7.57 times faster than NonIncBack for $goal_3$ and $goal_4$, respectively. When the previous analysis result is true, IncBack does not perform the second stage and hence the execution time of the second stage is 0. When a can_assign rule is deleted, IncBack is 14.84 and 28.1 times faster than NonIncBack for $goal_3$ and $goal_4$, respectively.

6 Related Work

A number of researchers investigated the problem of analyzing (a subset of) static ARBAC policies. In contrast, we consider changes to ARBAC policies. Below, we summarize their work. Li et al. [10] presented algorithms and complexity results for analysis of two restricted versions of ARBAC97 – AATU and AAR. This work did not consider negative preconditions in ARBAC policies. Schaad and Moffett [17] used the Alloy analyzer [7] to check separation of duty properties for ARBAC97. However, they did not consider preconditions in ARBAC policies. Sasturkar et al. [16] and Jha et al. [8] presented algorithms and complexity results for analysis of ARBAC policies subject to a variety of restrictions. Stoller et. al. [18] proposed the first fixed-parameter-tractable algorithms for analyzing ARBAC policies, which lays the groundwork for incremental analysis algorithms presented in this paper.

Crampton [2] showed undecidability of reachability analysis for RBAC, whose changes are controlled by commands consisting of pre-conditions and administrative operations; some commands are not expressible in the form allowed in ARBAC97 and some commands use administrative operations that change the ARBAC policy.

Prior works [6, 14, 9] have also considered changes to access control policies. However, the changes are not controlled by ARBAC, nor did these work consider changes to administrative policies. Fisler *et al.* [3] developed a change-impact analysis algorithm for RBAC policies based on binary decision diagram (BDD) by computing the semantic difference of two policies and checking properties of the difference. We consider changes to ARBAC policies, instead of RBAC policies. Furthermore, we also propose a lazy analysis algorithm.

Incremental computation has been studied in several areas, including deductive databases, logic programming, and program analysis. However, to the best of our knowledge, we are the first to develop the incremental algorithms for ARBAC policy analysis. Gupta [4] incrementally updated materialized views in databases by counting the number of derivations for each relation. Our approach is more efficient for analyzing ARBAC policies: we compute each transition only once; counting derivations would require determining all ways in which a transition can be computed. Gupta *et al.* [5] proposed a two-phase delete-rederive algorithm, which first deletes relations that depend on the deleted relation, and then rederives the deleted relations that have alternative derivations. Similar approaches were adapted in [13]. We avoid the rederivation phase by removing only those roles from the state for which all derivations have been invalidated. Lu *et al.* [11] proposed a Straight Delete algorithm (StDel) which eliminates the rederivation phase of delete-rederive algorithm. Direct application of StDel to ARBAC policy analysis would require storing all derivations for every state and every transition and, just as with counting, would be less efficient. Conway *et. al.* [1] developed algorithms to incrementally update control flow graphs of C programs. Since ARBAC has no control flow, their algorithms are not directly applicable to our problem. All aforementioned work, in contrast to our algorithms, computed the exact data structure. Further, none of them have proposed a lazy algorithm as we do.

7 Conclusion

This paper presents algorithms for incremental user-role reachability analysis of ARBAC policies. The incremental algorithms identify changes to the policy that may affect the analysis result and use the information computed in the previous analysis to update the result. We have evaluated our incremental algorithms using a set of randomly generated ARBAC policies and a set of operations that change the policies. Our experimental data show that our incremental algorithms outperform the non-incremental algorithm in terms of execution time. We will further optimize the incremental analysis algorithms. A promising optimization is not to perform operations, which do not affect the analysis result, on the graph. Such operations include operations that remove irrelevant roles from the graph and operations that change visible transitions to invisible transitions.

Acknowledgement. The authors thank Jian He, Jian Zhang and Arunpriya somasundaram for their contributions to the implementation.

References

1. C. Conway, K. Namjoshi, D. Dams, and S. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *Computer Aided Verification*, 2005.
2. J. Crampton. Authorizations and antichains, ph.d. thesis, university of london. 2002.
3. K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *International Conference on Software Engineering (ICSE)*, pages 196–205, 2005.
4. A. Gupta, D. Katiyar, and I. S. Mumick. Counting solutions to the view maintenance problem. In *Workshop on Deductive Databases*, pages 185–194, 1992.
5. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *International Conference on Management of Data*, pages 157–166, 1993.
6. M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.
7. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. pages 730–733, June 2000.
8. S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough. Towards formal verification of role-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 5(2), 2008.
9. S. Jha and T. Reps. Model-checking SPKI-SDSI. *Journal of Computer Security*, 12:317–353, 2004.
10. N. Li and M. V. Tripunitara. Security analysis in role-based access control. *ACM Transactions on Information and System Security*, 9(4):391–420, Nov. 2006.
11. J. Lu, G. Moerkotte, J. Schu, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views, 1995.
12. D. F. F. R. Sandhu and D. R. Kuhn. The NIST model for role based access control: Towards a unified standard. In *ACM SACMAT*, pages 47–63, 2000.
13. D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *In International Conference on Logic Programming*, pages 392–406, 2003.
14. R. Sandhu. The typed access matrix model. In *Proc. IEEE Symposium on Security and Privacy*, pages 122–136, 1992.
15. R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and Systems Security*, 2(1):105–135, 1999.
16. A. Sasturkar, P. Yang, S. D. Stoller, and C. Ramakrishnan. Policy analysis for administrative role based access control. In *19th IEEE Computer Security Foundations Workshop*, 2006.
17. A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Proc.of SACMAT*, pages 13–22, 2002.
18. S. Stoller, P. Yang, C. R. Ramakrishnan, and M. Gofman. Efficient policy analysis for administrative role based access control. In *ACM CCS*, pages 445–455, 2007.